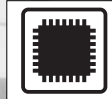
基礎から学ぶ Verilog HDL & FPGA 設計

第2回
4ビット加算器を設計しよう

中野浩嗣，伊藤靖朗

前回（本誌2007年4月号，pp.105-114）は，全加算器をVerilog HDLで設計し，シミュレーションや，FPGAボードへの回路のダウンロードと動作確認を行った．今回は，前回設計した全加算器を用いて，4ビット加算器を設計する．（編集部）

1．4ビット加算器の設計

前回設計した全加算器は，組み合わせ回路の一つです．組み合わせ回路とは，現在の入力にのみ依存して出力が決まる回路で，基本ゲート回路（NOT，AND，OR，XORなど）の組み合わせで設計することができます．今回は，もう少し複雑な組み合わせ回路に挑戦しましょう．

● assign文とalways文

前回は全加算器を設計するのにassign文を用いましたが，Verilog HDLでは，通常，assign文よりもalways文の方がよく用いられます．**リスト1**に，always文を用いた全加算器のVerilog HDL記述を示します．

リスト1　always文を用いた全加算器のVerilog HDL記述（fa.v）

```
 1  module fa(a, b, cin, s, cout);
 2
 3     input a, b, cin;
 4     output s, cout;
 5     reg s, cout;  ← レジスタ宣言
 6
 7     always @(a or b or cin)
 8     begin
 9       s = a ^ b ^ cin;
10       cout = (a & b) | (b & cin) | (cin & a);
11     end
12                          ↑ aかbかcinの値が変化
13  endmodule                 するたびに実行される
```

4行目までは，前回のassign文を用いた全加算器と同じです．5行目でreg文を使って変数sとcoutをレジスタ型変数として宣言しています．always文は，always @(...)という形で始まります．「...」の部分はイベント・リストと呼ばれ，ここで指定された変数などの値に変化があるたびに，後に続く文が実行されます．ここでイベント・リストは「a or b or cin」なので，入力信号のa，b，cinのいずれかの値が変化するたびに，後に続くbegin～end間の文（9行目と10行目の代入文）が実行されます．よって，変数sとcoutは常に正しい値をとることになります．

5行目のreg文は「レジスタ宣言」と呼ばれ，sとcoutがレジスタ型変数であることを宣言しています．ただし，このレジスタ型変数は，いわゆるレジスタ（論理回路で値を記憶するのに用いる）になるとは限りません．レジスタ型変数は，レジスタにも信号線にもなり得ます．実際，**リスト1**のsとcoutは信号線です．これがVerilog HDLのややこしい点なのですが，ともあれ，以下の点を覚えておくとよいでしょう．

- always文の中で用いられる代入文の左辺は，レジスタ宣言（reg）で宣言されたレジスタ型変数でなければならない．
- assign文の中で用いられる代入文の左辺は，ネット宣言（wire）で宣言されたネット型変数でなければならない．

KeyWord　Verilog HDL，FPGA，HDL，全加算器，イベント・リスト，レジスタ，モジュール・インスタンス化，オーバフロー，1の補数，2の補数，演算子

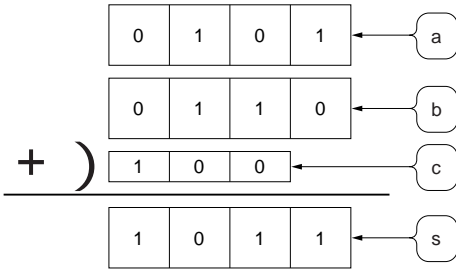



図1　4ビット加算器のイメージ
4ビットの2進数を二つ加算して，4ビットの値を出力する．けた上がりのためのビットも必要．

図2　4ビット加算器のブロック図
前回設計したモジュールfaを適切に接続すれば，4ビット加算器を作ることができる．

- レジスタ型変数は，値を保持するレジスタにも信号線にもなり得る．

レジスタ型変数がどのような場合に信号線になり，レジスタになるのかについては，次回以降で詳しく説明します．

● 4ビット加算器を3通りの方法で記述する

次に，4ビットの2進数を二つ足し合わせる4ビット加算器を設計してみましょう．4ビット加算器は二つの4ビットをポートaとbに入力し，その合計を4ビットでポートsに出力します(**図1**)．前回設計した全加算器を四つ並べて適切に接続することにより，4ビット加算器を作ることができます(**図2**)．

ここでは，Verilog HDLのさまざまな記述方法を知るために，以下の三つの方法で4ビット加算器を設計してみます．

- すべての信号線の論理を定義する方法(最も原始的)
- 既に設計したモジュールを部品として利用する方法(モジュール・インスタンス化)
- 算術演算子を用いる方法

● すべての信号線の論理を定義する方法

リスト2は，すべての信号線をassign文で定義することにより設計した4ビット加算器です．3行目のinput文で，aとbが，それぞれ4ビットの入力ポートであることを宣言しています．ポートのインデックスの範囲を[3:0]と指定しているので，a[3]，a[2]，a[1]，a[0]は，4ビットからなる入力ポートaの，上位ビットから並べた各ビットを表すことになります．

入力ポートと同様に，4行目のoutput文では，sが4ビットの出力ポートであることを宣言しています．

5行目のwire文では，cが3ビットのネット(信号線)であることを宣言しています．この3ビットの信号は，入力ポートと出力ポートのいずれでもないモジュール内の信号線となります．

7行目から13行目のassign文で，各信号線の接続を定義しています．信号線c[2]，c[1]，c[0]がassign文の左辺と右辺の両方にあり，これらの信号線が全加算器を接続しています．

リスト2　すべての信号線をassign文で定義することにより設計した4ビット加算器のVerilog HDL記述(adder4.v その1)

```
 1  module adder4(a, b ,s);
 2
 3     input [3:0] a, b;
 4     output [3:0] s;
 5     wire [2:0] c;
 6
 7     assign s[0] = a[0] ^ b[0];
 8     assign c[0] = a[0] & b[0];
 9     assign s[1] = a[1] ^ b[1] ^ c[0];
10     assign c[1] = (a[1] & b[1]) | (b [1] & c[0]) |
11                                    (c[0] & a[1]);
12     assign s[2] = a[2] ^ b[2] ^ c[1];
13     assign c[2] = (a[2] & b[2]) | (b [2] & c[1]) |
14                                    (c[1] & a[2]);
15     assign s[3] = a[3] ^ b[3] ^ c[2];

   endmodule
```
(7～8行目の注記：最下位ビット同士の加算)

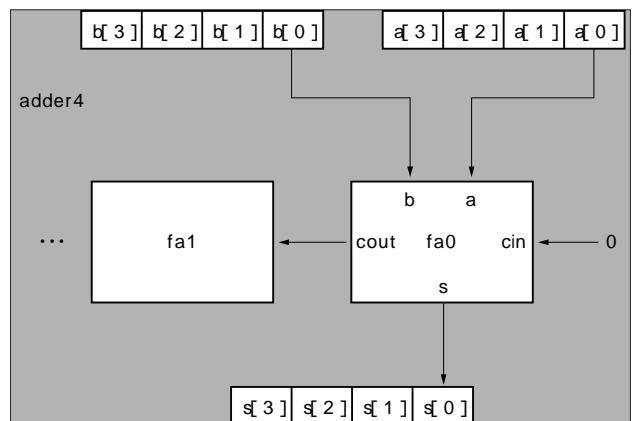
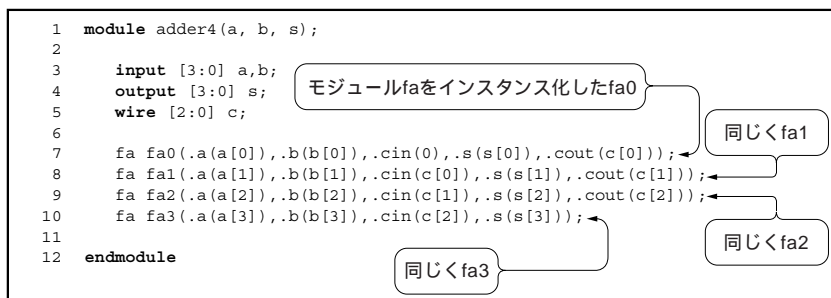
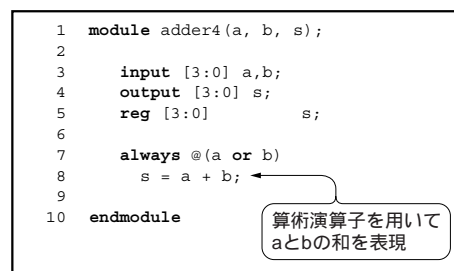



図3　モジュールadder4の信号線とモジュールfaの接続

リスト3　モジュールfaを用いた4ビット加算器のVerilog HDL記述（adder4.v その2）

```
 1  module adder4(a, b, s);
 2
 3     input [3:0] a,b;
 4     output [3:0] s;
 5     wire [2:0] c;
 6
 7     fa fa0(.a(a[0]),.b(b[0]),.cin(0),.s(s[0]),.cout(c[0]));
 8     fa fa1(.a(a[1]),.b(b[1]),.cin(c[0]),.s(s[1]),.cout(c[1]));
 9     fa fa2(.a(a[2]),.b(b[2]),.cin(c[1]),.s(s[2]),.cout(c[2]));
10     fa fa3(.a(a[3]),.b(b[3]),.cin(c[2]),.s(s[3]));
11
12  endmodule
```


リスト4　算術演算子を用いた4ビット加算器のVerilog HDL記述（adder4.v その3）

```
 1  module adder4(a, b, s);
 2
 3     input [3:0] a,b;
 4     output [3:0] s;
 5     reg [3:0]        s;
 6
 7     always @(a or b)
 8       s = a + b;
 9
10  endmodule
```


● **モジュール・インスタンス化を利用する方法**

既に設計したモジュールfaを利用して，4ビット加算器を作ってみましょう．このモジュールfaは，assign文を用いたものでもalways文を用いたものでもかまいません．

リスト3は，モジュールfaを用いた4ビット加算器のVerilog HDL記述です．7行目～10行目は，モジュールfaを四つ用い，それぞれにfa0，fa1，fa2，fa3という名前を付けることを宣言しています．このように，既に設計したモジュールを用いることを「モジュールのインスタンス化」と呼びます注1．

かっこの中は，各モジュールfaのポートと，モジュールadder4のネット（信号線）との接続を定義しています．例えば，7行目の.a(a[0])は，fa0の入力ポートaにモジュールadder4のネットa[0]を接続することを意味します（**図3**）．

また，.cin(0)は，fa0の入力ポートcinに常に0を書き込むことを意味します．さらに，.s(s[0])は，fa0の出力ポートsにモジュールadder4のネットs[0]を接続することになり，fa0の出力ポートsの値がs[0]に書き込まれることになります．このように，**図2**に示されているfa0，fa1，fa2，fa3間の接続が7行目～10行目で実現されていることを確認してみてください．

なお，モジュール・インスタンス化の際に，ポート・リストの順序に合わせて，接続するネットを記述する方法があります．例えば，**リスト3**の7行目の場合，モジュールfaのポート・リストはa，b，cin，s，coutの順なので，ポートを省略して，

```
fa fa0(a[0],b[0],0,s[0],c[0]);
```

と書くことができます．しかし，このような記述はバグの原因となりやすいので，本連載では，なるべくポート・リストを省略しない書き方をすることにします．

注1：モジュールのインスタンス化については第1回（本誌2007年4月号に掲載）にも説明した．インスタンス宣言の書式については，2007年4月号のp.111を参照のこと．

● **算術演算子を用いる方法**

算術演算子を用いることにより，前の二つの方法に比べて加算回路を非常に簡単に設計することができます．**リスト4**は，always文を使って，8行目でsにaとbの和を書き込んでいます．

ここで，+は算術演算子であり，aとbの合計が計算されます．従って，aとbのいずれかのビットの値が変化した場合にこの代入文が実行されます．

この例のように，代入文が一つの場合，begin～endは省略することができます．

また，**リスト4**はalways文を用いていますが，かわりにassign文を用いることができます．具体的には，5行目のレジスタ宣言（reg）を削除して，7行目と8行目を，

```
assign s = a + b;
```

で置き換えます．

2. 補数表現を使って正しい計算結果を得る

4ビットの加算器は，二つの4ビットのビット列を加算します．Verilog HDLでは，ビット列を数値として取り扱うとき，「符号なしの2進数表現」とみなします．つまり，“0000”，“0001”，…，“1111”は，0，1，…，15として取り扱われます（**表1**）．4ビット加算器adder4の出力は4ビットなので，加算結果が4ビットの範囲に収まるとき，つまり，0～15のとき，正しい計算結果となります．

ところが，15を超えるとき，つまり，全加算器fa3から

表1　4ビットのビット列に対する値(10進数で表した)

ビット列	符号なし2進数表現	符号付き2進数表現	1の補数表現	2の補数表現
0000	0	0	0	0
0001	1	1	1	1
0010	2	2	2	2
0011	3	3	3	3
0100	4	4	4	4
0101	5	5	5	5
0110	6	6	6	6
0111	7	7	7	7
1000	8	−0(=0)	−7	−8
1001	9	−1	−6	−7
1010	10	−2	−5	−6
1011	11	−3	−4	−5
1100	12	−4	−3	−4
1101	13	−5	−2	−3
1110	14	−6	−1	−2
1111	15	−7	−0(=0)	−1

けた上がりが生じるときは4ビットの範囲に収まらないので，正しい計算結果になりません．例えば，4ビット加算器による計算"0011"+"1110"="0001"は，10進数に直すと3＋14＝1となり，誤った結果になります．このような状況を，けたあふれ(オーバフロー)と言います．

● 符号付き2進数

CPUなどで計算を行う場合は，負の数が取り扱えないと不便です．そこで，ビット列の一部を負の数とみなすようにします．簡単に思いつくのは，最上位ビットを符号とみなす「符号付き2進数表現」です．

最上位ビットが1のときは負の数とみなし，下位3ビットがその絶対値であるとします(**表1**)．例えば，"1101"は，最上位ビットが1なので負の数であり，絶対値は下位3ビット"101"，つまり5なので，−5とみなします．

先に設計した4ビット加算器adder4では，ビット列を符号付き2進数表現とみなすと加算結果が正しくないことがあります．例えば，"0011"+"1110"="0001"は，3＋(−6)＝1となり，間違った結果になってしまいます．そこで，1の補数表現や2の補数表現を用います．

● 1の補数表現

1の補数とは，ビット列の各ビットを反転したものです．例えば，"0110"の1の補数は"1001"です．

1の補数表現では，1の補数を符号が反転したものと考えます．例えば，"0110"は符号なし2進数表現では6なので，その1の補数"1001"を−6とみなします．これによると，最上位ビットが1であるビット列は負の数であることになります．すると，**表1**に示すように，4ビット列に負の数を割り当てることができます．

1の補数表現では，絶対値が同じ正の数と負の数を加算すると全ビットが1になります．例えば，6＋(−6)＝0は"0110"+"1001"="1111"となります．"1111"は−0，つまり0なので，正しい加算結果になります．

ところが，1の補数表現では"0000"と"1111"の2通りのビット列が0となってしまいます．これはとても不都合です．例えば，二つのビット列が同じ値かどうかを判定する回路を設計するときに，"0000"と"1111"は同じ値である，と例外判定する必要があり，回路が複雑になってしまいます．

● 2の補数表現

ビット列の各ビットを反転し，1を加算したものを「2の補数」と呼びます．例えば，"0110"の2の補数は"1010"です．ほとんどのCPUでは，1の補数表現の欠点を克服した2の補数表現を用いています．

2の補数表現では，2の補数を符号が反転したものと考えます．例えば，"0110"は符号なし2進数表現では6なので，その2の補数"1010"を−6とみなします．

1の補数と同様に，最上位ビットが1のビット列は，負の数であるとします．すると，**表1**に示すように，4ビット列に負の数を割り当てることができます．

2の補数表現では，絶対値が同じ正の数と負の数を加算すると全ビットが0になり，正しい結果になっています．例えば，6＋(−6)＝0は"0110"+"1010"="0000"となり，結果は正しくなります．

4ビットの2の補数表現は−8～7の値をとりますが，計算結果がこの範囲に収まる限り，4ビット加算器は入出力ビット列を2の補数表現とみなしても正しい計算結果を出力します．

なお本連載は，ゴールとしてCPUを設計することを考えています．それを踏まえて，今後，ビット列が数値を表すときは，2の補数表現であるものとします．

ただしVerilog HDLでは，ビット列が「符号なしの2進数表現」として取り扱われ，各種演算が行われるので，この点を考慮して設計する必要があります．

3. 4ビット加算器をシミュレーションする

さてここからは，手を動かします．作成した4ビット加算器adder4.vを設計ツール（ISE WebPACK）に入力し，正しく動作することをシミュレーションで確認してみましょう．

設計ツールを起動すると，前回作成したプロジェクトが開きます（開いていない場合は，メニュー・バーから「File」→「Open Project...」を選択してプロジェクトを開く）．Sourcesウィンドウ内を右クリックし，「New Source...」を選択して「Verilog Module」を選択し，File nameを「adder4」として，ファイルadder4.vを作成します．

adder4.vに入力するVerilog HDL記述は，3通りの設計（**リスト2**～**リスト4**）のうち，どれでもかまいません．いずれの場合も結果は同じになります．

● テストベンチの作成

次に，テストベンチadder4_tb.vを作成します．作成方法は前回行ったfa_tb.vの作成と同様です．

具体的には．Sourcesウィンドウ内を右クリックし，「New Source...」を選択して「Verilog Module」を選択し，File nameを「adder4_tb.v」として，ファイルadder4_tb.vを作成します．Sourcesウィンドウにadder4_tb.vのテンプレートが表示されるので，これを編集して，adder4.vに対するテストベンチを作成します．

リスト5にテストベンチの例を示します．テストベンチなので，モジュールadder4_tbにポートはありません．

4行目で，入力のためのレジスタ型変数a，bを宣言します．5行目で出力のためのネット型変数sを宣言します．7行目では，モジュールadder4をインスタンス化しています．9行目から19行目では，入力aとbの値を設定しています．

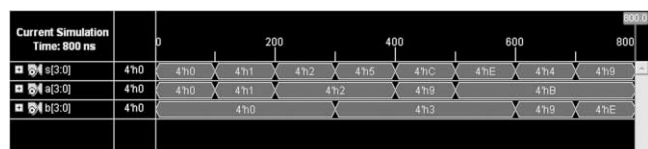
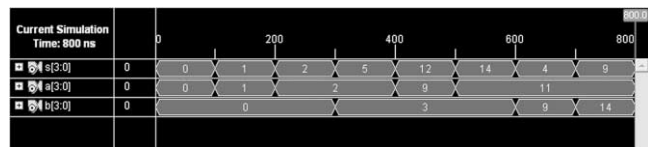
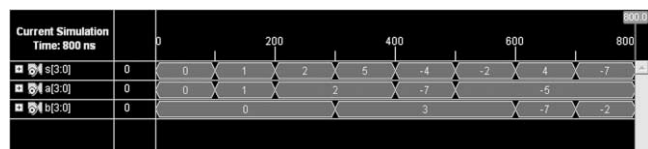
テストベンチを入力した後，構文チェックを行います．Sourcesウィンドウ上部のドロップダウン・リストから「Behavioral Simulation」を選択します．次に，Sourcesウィンドウのadder4_tb（adder4_tb.v）を選択し，Processesウィンドウの「Xilinx ISE Simulator」の階層を展開して，「Check Syntax」をダブルクリックします．

問題がなければ，Processesウィンドウの「Simulate Behavioral Model」をダブルクリックし，シミュレーションを行います．シミュレーション結果はワーク・スペースに表示されます．

シミュレーション波形において，4ビットの各ネットa, b，sの値は16進数で表示されます［**図4（a）**］．加算結果が正しいかどうかを確認するために，表示形式を変更してみましょう．

変更したい波形にマウス・カーソルを合わせ，右ボタンをクリックすると，**図5**のメニューが現れます．このメニューの「Decimal（Unsigned）」を選ぶと，ビット列を符号なし2進数表現とみなした場合の値を10進数で表示しま

リスト5　テストベンチ（adder4_tb.v）

```
 1  `timescale 1ns / 1ps
 2  module adder4_tb;
 3
 4  reg [3:0] a,b;
 5  wire [3:0] s;
 6
 7  adder4 adder4_0(.a(a),.b(b),.s(s));
 8
 9  initial begin
10      a = 4'b0000; b=4'b0000;
11      #100 a = 4'b0001;
12      #100 a = 4'b0010;
13      #100 b = 4'b0111;
14      #100 a = 4'b1101;
15      #100 a = 4'b1011;
16      #100 b = 4'b1001;
17      #100 b = 4'b1110;
18      #100 a = 4'b0000; b=4'b0000;
19  end
20
21  endmodule
```

（a）16進数表現

（b）符号なし2進数表現

（c）2の補数表現

図4　4ビット加算器adder4のシミュレーション波形

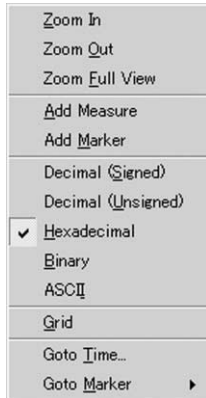



図5
波形の表示方式を変更する

す．「Decimal(Signed)」を選ぶと，2の補数表現とみなした場合の値を10進数で表示します．

符号なし2進数表現では，けたあふれのない(つまり，加算結果が0～15におさまる)限り，正しい結果になっています[**図4(b)**]．

2の補数表現でも，けたあふれのない(つまり，加算結果が－8～7におさまる)限り，正しい結果になっていることが確認できます[**図4(c)**]．

● Verilog HDLの演算子

Verilog HDLでは加算＋を含めて，さまざまな演算子が用意されています．Verilog HDLの主な演算子を**表2**に示します．一部を除いて，C言語の演算子をほぼ踏襲しています．乗算，除算，剰余は複雑な回路になるので，設計ツールによってはサポートしていない場合があります．

先に述べたように，Verilog HDLでは，ビット列は符号なしの2進表現として扱われる点に注意する必要があります．

CPUでは，負の数を扱うためにビット列を2の補数表現とみなして演算を行いたいので，Verilog HDLの演算子がそのまま使えないことがあります．4ビット加算器で確認したように，加算＋については，ビット列を2の補数表現とみなしても，正しい演算結果が得られます．減算も同様です．

実は乗算＊についても，ビット列を2の補数表現とみなしても正しい演算ができます．4ビットのビット列で確認してみましょう．乗算(－2)＊3は，2の補数表現では，“1110”＊“0011”となります．この4ビット列の乗算を符号なし2進表現とみなして計算すると，結果は“101010”となります．この下位4ビットを取り出すと“1010”であり，2の補数表現では－6なので，正しい計算結果になっています．よって，加算，減算，乗算に関しては，Verilog HDLの算術演算子をそのまま用いて，2の補数表現として演算を行うことができます．

表2　Verilog HDLの主な演算子

<table>
<tr><th colspan="2">算術演算子</th></tr>
<tr><td>+</td><td>加算</td></tr>
<tr><td>-</td><td>減算</td></tr>
<tr><td>*</td><td>乗算</td></tr>
<tr><td>/</td><td>除算</td></tr>
<tr><td>%</td><td>剰余算</td></tr>
<tr><td>-</td><td>2の補数(符号反転)</td></tr>
<tr><th colspan="2">ビットごとの演算子</th></tr>
<tr><td>~</td><td>1の補数(ビットごとの反転)</td></tr>
<tr><td>&</td><td>ビットごとの論理積</td></tr>
<tr><td>|</td><td>ビットごとの論理和</td></tr>
<tr><td>^</td><td>ビットごとの排他的論理和</td></tr>
<tr><td>~^</td><td>ビットごとの排他的論理和否定</td></tr>
<tr><th colspan="2">ビット・シフト演算子</th></tr>
<tr><td><<</td><td>左辺を右辺だけ左シフト</td></tr>
<tr><td>>></td><td>左辺を右辺だけ右シフト</td></tr>
<tr><th colspan="2">論理演算子</th></tr>
<tr><td>!</td><td>論理否定</td></tr>
<tr><td>&&</td><td>論理積</td></tr>
<tr><td>||</td><td>論理和</td></tr>
</table>

<table>
<tr><th colspan="2">関係演算子</th></tr>
<tr><td>==</td><td>左辺と右辺は等しい</td></tr>
<tr><td>!=</td><td>左辺と右辺は等しくない</td></tr>
<tr><td>>=</td><td>左辺は右辺以上</td></tr>
<tr><td><=</td><td>左辺は右辺以下</td></tr>
<tr><td>></td><td>左辺は右辺より大きい</td></tr>
<tr><td><</td><td>左辺は右辺より小さい</td></tr>
<tr><th colspan="2">リダクション演算子</th></tr>
<tr><td>&</td><td>全ビットの論理積</td></tr>
<tr><td>~&</td><td>全ビットの論理積否定</td></tr>
<tr><td>|</td><td>全ビットの論理和</td></tr>
<tr><td>~|</td><td>全ビットの論理和否定</td></tr>
<tr><td>^</td><td>全ビットの排他的論理和</td></tr>
<tr><td>~^</td><td>全ビットの排他的論理和否定</td></tr>
<tr><th colspan="2">条件演算子</th></tr>
<tr><td>?:</td><td>条件?真の場合:偽の場合</td></tr>
<tr><th colspan="2">連結演算子</th></tr>
<tr><td>{,}</td><td>ビット列の連結</td></tr>
<tr><th colspan="2">反復演算子</th></tr>
<tr><td>{{}}</td><td>ビット列の繰り返し</td></tr>
</table>

問題は，関係演算子の大小比較です．例えば，4ビットのビット列“1101”と“0001”は，符号なしの2進数と見なすと13＞1ですが，2の補数表現では－3＜1となり，逆の結果となります．2の補数表現として正しい結果にするためには，ある工夫が必要です．これについては次回で触れることとします．

次回は，マルチプレクサと算術論理演算回路(arithmetic and logic unit：ALU)を設計します．

なかの・こうじ
いとう・やすあき
広島大学大学院 工学研究科

<筆者プロフィール>
中野浩嗣．1992年，大阪大学大学院 博士後期課程修了．工学博士．一つの民間企業，二つの大学を経て，2003年より広島大学 教授．
伊藤靖朗．2003年，北陸先端科学技術大学院大学 博士前期課程修了．現在，広島大学 助教．